

Benchmarking for Bayesian Reinforcement Learning

Michaël Castronovo

M.CASTRONOVO@ULG.AC.BE

*Montefiore Institute
University of Liège
B-4000 Liège, Belgium*

Damien Ernst

DERNST@ULG.AC.BE

*Montefiore Institute
University of Liège
B-4000 Liège, Belgium*

Adrien Couëtoux

ACOUETOUX@ULG.AC.BE

*Montefiore Institute
University of Liège
B-4000 Liège, Belgium*

Raphaël Fonteneau

RAPHAEL.FONTENEAU@ULG.AC.BE

*Montefiore Institute
University of Liège
B-4000 Liège, Belgium*

Abstract

In the Bayesian Reinforcement Learning (BRL) setting, agents try to maximise the collected rewards while interacting with their environment while using some prior knowledge that is accessed beforehand. Many BRL algorithms have already been proposed, but even though a few toy examples exist in the literature, there are still no extensive or rigorous benchmarks to compare them. The paper addresses this problem, and provides a new BRL comparison methodology along with the corresponding open source library. In this methodology, a comparison criterion that measures the performance of algorithms on large sets of Markov Decision Processes (MDPs) drawn from some probability distributions is defined. In order to enable the comparison of non-anytime algorithms, our methodology also includes a detailed analysis of the computation time requirement of each algorithm. Our library is released with all source code and documentation: it includes three test problems, each of which has two different prior distributions, and seven state-of-the-art RL algorithms. Finally, our library is illustrated by comparing all the available algorithms and the results are discussed.

Keywords: Bayesian Reinforcement Learning, Benchmarking, BBRL library, Offline Learning, Reinforcement Learning

1. Introduction

Reinforcement Learning (RL) agents aim to maximise collected rewards by interacting over a certain period of time in initially unknown environments. Actions that yield the highest performance according to the current knowledge of the environment and those that maximise the gathering of new knowledge on the environment may not be the same. This is the dilemma known as Exploration/Exploitation (E/E). In such a context, using prior knowledge of the environment is extremely valuable, since it can help guide the decision-

making process in order to reduce the time spent on exploration. Model-based Bayesian Reinforcement Learning (BRL) (Dearden et al. (1999); Strens (2000)) specifically targets RL problems for which such a prior knowledge is encoded in the form of a probability distribution (the “prior”) over possible models of the environment. As the agent interacts with the actual model, this probability distribution is updated according to the Bayes rule into what is known as “posterior distribution”. The BRL process may be divided into two learning phases: the offline learning phase refers to the phase when the prior knowledge is used to warm-up the agent for its future interactions with the real model. The online learning phase, on the other hand, refers to the actual interactions between the agent and the model. In many applications, interacting with the actual environment may be very costly (e.g. medical experiments). In such cases, the experiments made during the online learning phase are likely to be much more expensive than those performed during the offline learning phase.

In this paper, we investigate how the way BRL algorithms use the offline learning phase may impact online performances. To properly compare Bayesian algorithms, the first comprehensive BRL benchmarking protocol is designed, following the foundations of Castronovo et al. (2014). “Comprehensive BRL benchmark” refers to a tool which assesses the performance of BRL algorithms over a large set of problems that are actually drawn according to a prior distribution. In previous papers addressing BRL, authors usually validate their algorithm by testing it on a few test problems, defined by a small set of predefined MDPs. For instance, BAMCP (Guez et al. (2012)), SBOSS (Castro and Precup (2010)), and BFS3 (Asmuth and Littman (2011)) are all validated on a fixed number of MDPs. In their validation process, the authors select a few BRL tasks, for which they choose one arbitrary transition function, which defines the corresponding MDP. Then, they define one prior distribution compliant with the transition function. This type of benchmarking is problematic in the sense that the authors actually know the hidden transition function of each test case. It also creates an implicit incentive to over-fit their approach to a few specific transition functions, which should be completely unknown before interacting with the model. In this paper, we compare BRL algorithms in several different tasks. In each task, the real transition function is defined using a random distribution, instead of being arbitrarily fixed. Each algorithm is thus tested on an infinitely large number of MDPs, for *each* test case. To perform our experiments, we developed the *BBRL* library, whose objective is to also provide other researchers with our benchmarking tool.

This paper is organised as follows: Section 2 presents the problem statement. Section 3 formally defines the experimental protocol designed for this paper. Section 4 briefly presents the library. Section 5 shows a detailed application of our protocol, comparing several well-know BRL algorithms on three different benchmarks. Section 6 concludes the study.

2. Problem Statement

This section is dedicated to the formalisation of the different tools and concepts discussed in this paper.

2.1 Reinforcement Learning

Let $M = (X, U, f(\cdot), \rho_M, p_{M,0}(\cdot), \gamma)$ be a given unknown MDP, where $X = \{x^{(1)}, \dots, x^{(n_X)}\}$ denotes its finite state space and $U = \{u^{(1)}, \dots, u^{(n_U)}\}$ refers to its finite action space. When the MDP is in state x_t at time t and action u_t is selected, the agent moves instantaneously to a next state x_{t+1} with a probability of $P(x_{t+1}|x_t, u_t) = f(x_t, u_t, x_{t+1})$. An instantaneous deterministic, bounded reward $r_t = \rho_M(x_t, u_t, x_{t+1}) \in [R_{\min}, R_{\max}]$ is observed.

Let $h_t = (x_0, u_0, r_0, x_1, \dots, x_{t-1}, u_{t-1}, r_{t-1}, x_t) \in H$ denote the history observed until time t . An E/E strategy is a stochastic policy π which, given the current state x_t , returns an action $u_t \sim \pi(h_t)$. Given a probability distribution over initial states $p_{M,0}(\cdot)$, the expected return of a given E/E strategy π with respect to the MDP M can be defined as follows:

$$J_M^\pi = \mathbb{E}_{x_0 \sim p_{M,0}(\cdot)} [\mathcal{R}_M^\pi(x_0)],$$

where $\mathcal{R}_M^\pi(x_0)$ is the stochastic sum of discounted rewards received when applying the policy π , starting from an initial state x_0 :

$$\mathcal{R}_M^\pi(x_0) = \sum_{t=0}^{+\infty} \gamma^t r_t.$$

RL aims to learn the behaviour that maximises J_M^π , i.e. learning a policy π^* defined as follows:

$$\pi^* \in \arg \max_{\pi} J_M^\pi.$$

2.2 Prior Knowledge

In this paper, the actual MDP is assumed to be initially unknown. Model-based Bayesian Reinforcement Learning (BRL) proposes to the model the uncertainty, using a probability distribution $p_{\mathcal{M}}^0(\cdot)$ over a set of candidate MDPs \mathcal{M} . Such a probability distribution is called a prior distribution and can be used to encode specific prior knowledge available before interaction. Given a prior distribution $p_{\mathcal{M}}^0(\cdot)$, the expected return of a given E/E strategy π is defined as:

$$\mathfrak{J}_{p_{\mathcal{M}}^0(\cdot)}^\pi = \mathbb{E}_{M \sim p_{\mathcal{M}}^0(\cdot)} [J_M^\pi],$$

In the BRL framework, the goal is to maximise $\mathfrak{J}_{p_{\mathcal{M}}^0(\cdot)}^\pi$, by finding π^* , which is called ‘‘Bayesian optimal policy’’ and defined as follows:

$$\pi^* \in \arg \max_{\pi} \mathfrak{J}_{p_{\mathcal{M}}^0(\cdot)}^\pi.$$

2.3 Computation time characterisation

Most BRL algorithms rely on some properties which, given sufficient computation time, ensure that their agents will converge to an optimal behaviour. However, it is not clear to know beforehand whether an algorithm will satisfy fixed computation time constraints while providing good performances.

The parameterisation of the algorithms makes the selection even more complex. Most BRL algorithms depend on parameters (number of transitions simulated at each iteration, etc.) which, in some way, can affect the computation time. In addition, for one given algorithm and fixed parameters, the computation time often varies from one simulation to another. These features make it nearly impossible to compare BRL algorithms under strict computation time constraints. In this paper, to address this problem, algorithms are run with multiple choices of parameters, and we analyse their time performance a posteriori.

Furthermore, a distinction between the offline and online computation time is made. Offline computation time corresponds to the moment when the agent is able to exploit its prior knowledge, but cannot interact with the MDP yet. One can see it as the time given to take the first decision. In most algorithms concerned in this paper, this phase is generally used to initialise some data structure. On the other hand, online computation time corresponds to the time consumed by an algorithm for taking each decision.

There are many ways to characterise algorithms based on their computation time. One can compare them based on the average time needed per step or on the offline computation time alone. To remain flexible, for each run of each algorithm, we store its computation times $(B_i)_{-1 \leq i}$, with i indexing the time step, and B_{-1} the offline learning time. Then a feature function $\phi((B_i)_{-1 \leq i})$ is extracted from this data. This function is used as a metric to characterise and discriminate algorithms based on their time requirements.

In our protocol, which is detailed in the next section, two types of characterisation are used. For a set of experiments, algorithms are classified based on their offline computation time only, i.e. we use $\phi((B_i)_{-1 \leq i}) = B_{-1}$. Afterwards, the constraint is defined as $\phi((B_i)_{-1 \leq i}) \leq K$, $K > 0$ in case it is required to only compare the algorithms that have an offline computation time lower than K .

For another set of experiments, algorithms are separated according to their empirical average online computation time. In this case, $\phi((B_i)_{-1 \leq i}) = \frac{1}{n} \sum_{0 \leq i < n} B_i$. Algorithms can then be classified based on whether or not they respect the constraint $\phi((B_i)_{-1 \leq i}) \leq K$, $K > 0$.

This formalisation could be used for any other computation time characterisation. For example, one could want to analyse algorithms based on the longest computation time of a trajectory, and define $\phi((B_i)_{-1 \leq i}) = \max_{-1 \leq i} B_i$.

3. A new Bayesian Reinforcement Learning benchmark protocol

3.1 A comparison criterion for BRL

In this paper, a real Bayesian evaluation is proposed, in the sense that the different algorithms are compared on a large set of problems drawn according to a test probability distribution. This is in contrast with the Bayesian literature (Guez et al. (2012); Castro and Precup (2010); Asmuth and Littman (2011)), where authors pick a fixed number of MDPs on which they evaluate their algorithm.

Our criterion to compare algorithms is to measure their average rewards against a given random distribution of MDPs, using another distribution of MDPs as a prior knowledge. In our experimental protocol, an experiment is defined by a prior distribution $p_{\mathcal{M}}^0(\cdot)$ and a test distribution $p_{\mathcal{M}}(\cdot)$. Both are random distributions over the set of possible MDPs, not stochastic transition functions. To illustrate the difference, let us take an example. Let

(x, u, x') be a transition. Given a transition function $f : X \times U \times X \rightarrow [0; 1]$, $f(x, u, x')$ is the probability of observing x' if we chose u in x . In this paper, this function f is assumed to be the only unknown part of the MDP that the agent faces. Given a certain test case, f corresponds to a unique MDP $M \in \mathcal{M}$. A Bayesian learning problem is then defined by a probability distribution over a set \mathcal{M} of possible MDPs. We call it a test distribution, and denote it $p_{\mathcal{M}}(\cdot)$. Prior knowledge can then be encoded as another distribution over \mathcal{M} , and denoted $p_{\mathcal{M}}^0(\cdot)$. We call “accurate” a prior which is identical to the test distribution ($p_{\mathcal{M}}^0(\cdot) = p_{\mathcal{M}}(\cdot)$), and we call “inaccurate” a prior which is different ($p_{\mathcal{M}}^0(\cdot) \neq p_{\mathcal{M}}(\cdot)$).

In previous Bayesian literature, authors select a fixed number of MDPs M_1, \dots, M_n , train and test their algorithm on them. Doing so does not guarantee any generalisation capabilities. To solve this problem, a protocol that allows rigorous comparison of BRL algorithms is designed. Training and test data are separated, and can even be generated from different distributions (in what we call the inaccurate case).

More precisely, our protocol can be described as follows: Each algorithm is first trained on the prior distribution. Then, their performances are evaluated by estimating the expectation of the discounted sum of rewards, when they are facing MDPs drawn from the test distribution. Let $\mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)}$ be this value:

$$\mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)} = \mathbb{E}_{M \sim p_{\mathcal{M}}} \left[J_M^{\pi(p_{\mathcal{M}}^0)} \right],$$

where $\pi(p_{\mathcal{M}}^0)$ is the algorithm π trained offline on $p_{\mathcal{M}}^0$. In our Bayesian RL setting, we want to find the algorithm π^* which maximises $\mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)}$ for the $\langle p_{\mathcal{M}}^0, p_{\mathcal{M}} \rangle$ experiment:

$$\pi^* \in \arg \max_{\pi} \mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)}.$$

In addition to the performance criterion, we also measure the empirical computation time. In practice, all problems are subject to time constraints. Hence, it is important to take this parameter into account when comparing different algorithms.

3.2 The experimental protocol

In practice, we can only sample a finite number of trajectories, and must rely on estimators to compare algorithms. In this section our experimental protocol is described, which is based on our comparison criterion for BRL and provides a detailed computation time analysis.

An experiment is defined by (i) a prior distribution $p_{\mathcal{M}}^0$ and (ii) a test distribution $p_{\mathcal{M}}$. Given these, an agent is evaluated π as follows:

1. Train π offline on $p_{\mathcal{M}}^0$.
2. Sample N MDPs from the test distribution $p_{\mathcal{M}}$.
3. For each sampled MDP M , compute estimate $\bar{J}_M^{\pi(p_{\mathcal{M}}^0)}$ of $J_M^{\pi(p_{\mathcal{M}}^0)}$.
4. Use these values to compute an estimate $\tilde{\mathfrak{J}}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)}$.

To estimate $J_M^{\pi(p_{\mathcal{M}}^0)}$, the expected return of agent π trained offline on $p_{\mathcal{M}}^0$, one trajectory is sampled on the MDP M , and the cumulated return is computed $\bar{J}_{M_i}^{\pi(p_{\mathcal{M}}^0)} = \mathcal{R}_M^{\pi(p_{\mathcal{M}}^0)}(x_0)$.

To estimate this return, each trajectory is truncated after T steps. Therefore, given an MDP M and its initial state x_0 , we observe $\bar{\mathcal{R}}_M^{\pi(p_{\mathcal{M}}^0)}(x_0)$, an approximation of $\mathcal{R}_M^{\pi(p_{\mathcal{M}}^0)}(x_0)$:

$$\bar{\mathcal{R}}_M^{\pi(p_{\mathcal{M}}^0)}(x_0) = \sum_{t=0}^T \gamma^t r_t.$$

If R_{max} denotes the maximal instantaneous reward an agent can receive when interacting with an MDP drawn from $p_{\mathcal{M}}$, then choosing T as guarantees the approximation error is bounded by $\epsilon > 0$:

$$T = \left\lceil \frac{\log(\epsilon \times \frac{(1-\gamma)}{R_{max}})}{\log \gamma} \right\rceil.$$

$\epsilon = 0.01$ is set for all experiments, as a compromise between measurement accuracy and computation time.

Finally, to estimate our comparison criterion $\mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)}$, the empirical average of the algorithm performance is computed over N different MDPs, sampled from $p_{\mathcal{M}}$:

$$\mathfrak{J}_{p_{\mathcal{M}}}^{\pi(p_{\mathcal{M}}^0)} = \frac{1}{N} \sum_{0 \leq i < N} \bar{J}_{M_i}^{\pi(p_{\mathcal{M}}^0)} = \frac{1}{N} \sum_{0 \leq i < N} \bar{\mathcal{R}}_{M_i}^{\pi(p_{\mathcal{M}}^0)}(x_0) \quad (1)$$

For each agent π , we retrieve $\mu_{\pi} = \bar{J}_M^{\pi}$ and σ_{π} , the empirical mean and standard deviation of the results observed respectively. This gives us the following statistical confidence interval at 95% for J_M^{π} :

$$J_M^{\pi} \in \left[\bar{J}_M^{\pi} - \frac{2\sigma_{\pi}}{N}; \bar{J}_M^{\pi} + \frac{2\sigma_{\pi}}{N} \right].$$

The values reported in the following figures and tables are estimations of the interval within which J_M^{π} is, with probability 0.95.

As introduced in Section 2.3, in our methodology, a function ϕ of computation times is used to classify algorithms based on their time performance. The choice of ϕ depends on the type of time constraints that are the most important to the user. In this paper, we reflect this by showing three different ways to choose ϕ . These three choices lead to three different ways to look at the results and compare algorithms. The first one is to classify algorithms based on their offline computation time, the second one is to classify them based on the algorithms average online computation time. The third is a combination of the first two choices of ϕ , that we denote $\phi_{off}((B_i)_{-1 \leq i}) = B_{-1}$ and $\phi_{on}((B_i)_{-1 \leq i}) = \frac{1}{n} \sum_{0 \leq i < n} B_i$. The objective is that for each pair of constraints $\phi_{off}((B_i)_{-1 \leq i}) < K_1$ and $\phi_{on}((B_i)_{-1 \leq i}) < K_2$, $K_1, K_2 > 0$, we want to identify the best algorithms that respect these constraints. In order to achieve this: (i) All agents that do not satisfy the constraints are discarded; (ii) for each algorithm, the agent leading to the best performance in average is selected; (iii) we build the list of agents whose performances are not significantly different¹.

1. A paired sampled Z-test with a confidence level of 95% has been used to determine when two agents are statistically equivalent (more details in Appendix C).

The results will help us to identify, for each experiment, the most suitable algorithm(s) depending on the constraints the agents must satisfy. This protocol is an extension of the one presented in Castronovo et al. (2014).

4. BBRL library

*BBRL*² is a C++ open-source library for Bayesian Reinforcement Learning (discrete state/action spaces). This library provides high-level features, while remaining as flexible and documented as possible to address the needs of any researcher of this field. To this end, we developed a complete command-line interface, along with a comprehensive website:

<https://github.com/mcastron/BBRL>

BBRL focuses on the core operations required to apply the comparison benchmark presented in this paper. To do a complete experiment with the BBRL library, follow these five steps:

1. We create a test and a prior distribution. Those distributions are represented by Flat Dirichlet Multinomial distributions (FDM), parameterised by a state space X , an action space U , a vector of parameters θ , and reward function ρ . For more information about the FDM distributions, check Section 5.2.

```
./BBRL-DDS --mdp_distrib_generation \
--name <name> \
--short_name <short name> \
--n_states <nX> --n_actions <nU> \
--ini_state <x0> \
--transition_weights \
<θ(1)> ... <θ(nXnUnX)> \
--reward_type "RT_CONSTANT" \
--reward_means \
<ρ(x(1), u(1), x(1))> ... <ρ(x(nX), u(nU), x(nX))> \
--output <output file>
```

A distribution file is created.

2. We create an experiment. An experiment is defined by a set of N MDPs, drawn from a test distribution defined in a *distribution file*, a discount factor γ and a horizon limit T .

```
./BBRL-DDS --new_experiment \
--name <name> \
--mdp_distribution "DirMultiDistribution" \
--mdp_distribution_file <distribution file> \
--n_mdps <N> --n_simulations_per_mdp 1 \
--discount_factor <γ> --horizon_limit <T> \
```

2. BBRL stands for **B**enchmarking tools for **B**ayesian **R**einforcement **L**earning.

```
--compress_output \
--output <output file>
```

An experiment file is created and can be used to conduct the same experiment for several agents.

3. We create an agent. An agent is defined by an algorithm alg , a set of parameters ψ , and a prior distribution defined in a *distribution file*, on which the created agent will be trained.

```
./BBRL-DDS --offline_learning \
            --agent <alg> [<parameters  $\psi$ >]\
            --mdp_distribution "DirMultiDistribution" \
            --mdp_distribution_file <distribution file> \
            --output <output file>
```

An agent file is created. The file also stores the computation time observed during the offline training phase.

4. We run the experiment. We need to provide an *experiment file*, an algorithm alg and an *agent file*.

```
./BBRL-DDS --run_experiment \
            --experiment \
            --experiment_file <experiment file> \
            --agent <alg> \
            --agent_file <agent file> \
            --n_threads 1 \
            --compress_output \
            --safe_simulations \
            --refresh_frequency 60 \
            --backup_frequency 900 \
            --output <output file>
```

A *result file* is created. This file contains a set of all transitions encountered during each trajectory. Additionally, the computation times we observed are also stored in this file. It is often impossible to measure precisely the computation time of a single decision. This is why only the computation time of each trajectory is reported in this file.

5. Our results are exported. After each experiment has been performed, a set of K *result files* is obtained. We need to provide all *agent files* and *result files* to export the data.

```
./BBRL-export --agent <alg(1)> \
              --agent_file <agent file #1> \
              --experiment \
              --experiment_file <result file #1> \
```



```

...
--agent <alg(K)> \
  --agent_file <agent file #K> \
--experiment \
  --experiment_file <result file #K>

```

BBRL will sort the data automatically and produce several files for each experiment.

- A graph comparing offline computation cost w.r.t. performance;
- A graph comparing online computation cost w.r.t. performance;
- A graph where the X-axis represents the offline time bound, while the Y-axis represents the online time bound. A point of the space corresponds to set of bounds. An algorithm is associated to a point of the space if its best agent, satisfying the constraints, is among the best ones when compared to the others;
- A table reporting the results of each agent.

BBRL will also produce a report file in \LaTeX gathering the 3 graphs and the table for each experiment.

More than 2.000 commands have to be entered in order to reproduce the results of this paper. We decided to provide several *Lua* script in order to simplify the process. By completing some configuration files, the user can define the agents, the possible values of their parameters and the experiments to conduct.

```

local agents =
{
  -- e-Greedy
  {
    name = "EGreedyAgent",
    params =
    {
      {
        opt = "--epsilon",
        values =
        {
          0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
        }
      }
    },
    olOptions = { "--compress_output" },
    memory = { ol = "1000M", re = "1000M" },
    duration = { ol = "01:00:00", re = "01:00:00" }
  },
  ...
}

```

Figure 1: Example of a configuration file for the agents.

```

local experiments =
{
  {
    prior = "GC",    priorFile = "GC-distrib.dat",
    exp   = "GC",    testFile  = "GC-distrib.dat",
    N = 500, gamma = 0.95, T = 250
  },
  ...
}
    
```

Figure 2: Example of a configuration file for the experiments.

Those configuration files are then used by a script called `make_scripts.sh`, included within the library, whose purpose is to generate four other scripts:

- `0-init.sh`
Create the experiment files, and create the formulas sets required by OPPS agents.
- `1-ol.sh`
Create the agents and train them on the prior distribution(s).
- `2-re.sh`
Run all the experiments.
- `3-export.sh`
Generate the \LaTeX reports.

Due to the high computation power required, we made those scripts compatible with workload managers such as SLURM. In this case, each cluster should provide the same amount of CPU power in order to get consistent time measurements. To sum up, when the configuration files are completed correctly, one can start the whole process by executing the four scripts, and retrieve the results in nice \LaTeX reports.

It is worth noting that there is no computation budget given to the agents. This is due to the diversity of the algorithms implemented. No algorithm is “anytime” natively, in the sense that we cannot stop the computation at any time and receive an answer from the agent instantly. Strictly speaking, it is possible to develop an anytime version of some of the algorithms considered in *BBRL*. However, we made the choice to stay as close as possible to the original algorithms proposed in their respective papers for reasons of fairness. In consequence, although computation time is a central parameter in our problem statement, it is never explicitly given to the agents. We instead let each agent run as long as necessary and analyse the time elapsed afterwards.

Another point which needs to be discussed is the impact of the implementation of an algorithm on the comparison results. For each algorithm, many implementations are possible, some being better than others. Even though we did our best to provide the best possible implementations, *BBRL* does not compare algorithms but rather the implementations of each algorithms. Note that this issue mainly concerns small problems, since the complexity of the algorithms is preserved.

5. Illustration

This section presents an illustration of the protocol presented in Section 3. We first describe the algorithms considered for the comparison in Section 5.1, followed by a description of the benchmarks in Section 5.2. Section 5.3 shows and analyses the results obtained.

5.1 Compared algorithms

In this section, we present the list of the algorithms considered in this study. The pseudo-code of each algorithm can be found in Appendix A. For each algorithm, a list of “reasonable” values is provided to test each of their parameters. When an algorithm has more than one parameter, all possible parameter combinations are tested.

5.1.1 RANDOM

At each time-step t , the action u_t is drawn uniformly from U .

5.1.2 ϵ -GREEDY

The ϵ -Greedy agent maintains an approximation of the current MDP and computes, at each time-step, its associated Q-function. The selected action is either selected randomly (with a probability of ϵ ($1 \geq \epsilon \geq 0$), or greedily (with a probability of $1 - \epsilon$) with respect to the approximated model.

Tested values:

- $\epsilon \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

5.1.3 SOFT-MAX

The Soft-max agent maintains an approximation of the current MDP and computes, at each time-step, its associated Q-function. The selected action is selected randomly, where the probability to draw an action u is proportional to $Q(x_t, u)$. The temperature parameter τ allows to control the impact of the Q-function on these probabilities ($\tau \rightarrow 0^+$: greedy selection; $\tau \rightarrow +\infty$: random selection).

Tested values:

- $\tau \in \{0.05, 0.10, 0.20, 0.33, 0.50, 1.0, 2.0, 3.0, 5.0, 25.0\}$.

5.1.4 OPPS

Given a prior distribution $p_{\mathcal{M}}^0(\cdot)$ and an E/E strategy space \mathcal{S} (either discrete or continuous), the Offline, Prior-based Policy Search algorithm (OPPS) identifies a strategy $\pi^* \in \mathcal{S}$ which maximises the expected discounted sum of returns over MDPs drawn from the prior.

The OPPS for Discrete Strategy spaces algorithm (OPPS-DS) (Castronovo et al. (2012, 2014)) formalises the strategy selection problem as a k -armed bandit problem, where $k =$

$|\mathcal{S}|$. Pulling an arm amounts to draw an MDP from $p_{\mathcal{M}}^0(\cdot)$, and play the E/E strategy associated to this arm on it for one single trajectory. The discounted sum of returns observed is the return of this arm. This multi-armed bandit problem has been solved by using the UCB1 algorithm (Auer et al. (2002); Audibert et al. (2007)). The time budget is defined by a variable β , corresponding to the total number of draws performed by the UCB1.

The E/E strategies considered by Castronovo et. al are index-based strategies, where the index is generated by evaluating a small formula. A formula is a mathematical expression, combining specific features (Q-functions of different models) by using standard mathematical operators (addition, subtraction, logarithm, etc.). The discrete E/E strategy space is the set of all formulas which can be built by combining at most n features/operators (such a set is denoted by \mathbb{F}_n).

OPPS-DS does not come with any guarantee. However, the UCB1 bandit algorithm used to identify the best E/E strategy within the set of strategies provides statistical guarantees that the best E/E strategies are identified with high probability after a certain budget of experiments. However, it is not clear that the best strategy of the E/E strategy space considered yields any high-performance strategy regardless the problem.

Tested values:

- $\mathcal{S} \in \{\mathbb{F}_2, \mathbb{F}_3, \mathbb{F}_4, \mathbb{F}_5, \mathbb{F}_6\}^3$,
- $\beta \in \{50, 500, 1250, 2500, 5000, 10000, 100000, 1000000\}$.

5.1.5 BAMCP

Bayes-adaptive Monte Carlo Planning (BAMCP) (Guez et al. (2012)) is an evolution of the Upper Confidence Tree (UCT) algorithm (Kocsis and Szepesvári (2006)), where each transition is sampled according to the history of observed transitions. The principle of this algorithm is to adapt the UCT principle for planning in a Bayes-adaptive MDP, also called the belief-augmented MDP, which is an MDP obtained when considering augmented states made of the concatenation of the actual state and the posterior. The BAMCP algorithm is made computationally tractable by using a sparse sampling strategy, which avoids sampling a model from the posterior distribution at every node of the planification tree. Note that the BAMCP also comes with theoretical guarantees of convergence towards Bayesian optimality.

In practice, the BAMCP relies on two parameters: (i) Parameter K which defines the number of nodes created at each time-step, and (ii) Parameter *depth* which defines the depth of the tree from the root.

Tested values:

- $K \in \{1, 500, 1250, 2500, 5000, 10000, 25000\}$,
- $depth \in \{15, 25, 50\}$.

3. The number of arms k is always equal to the number of strategies in the given set. For your information: $|\mathbb{F}_2| = 12$, $|\mathbb{F}_3| = 43$, $|\mathbb{F}_4| = 226$, $|\mathbb{F}_5| = 1210$, $|\mathbb{F}_6| = 7407$

5.1.6 BFS3

The Bayesian Forward Search Sparse Sampling (BFS3) (Asmuth and Littman (2011)) is a Bayesian RL algorithm whose principle is to apply the principle of the FSSS (Forward Search Sparse Sampling, see Kearns et al. (2002)) algorithm to belief-augmented MDPs. It first samples one model from the posterior, which is then used to sample transitions. The algorithm then relies on lower and upper bounds on the value of each augmented state to prune the search space. The authors also show that BFS3 converges towards Bayes-optimality as the number of samples increases.

In practice, the parameters of BFS3 are used to control how much computational power is allowed. The parameter K defines the number of nodes to develop at each time-step, C defines the branching factor of the tree and $depth$ controls its maximal depth.

Tested values:

- $K \in \{1, 500, 1250, 2500, 5000, 10000\}$,
- $C \in \{2, 5, 10, 15\}$,
- $depth \in \{15, 25, 50\}$.

5.1.7 SBOSS

The Smarter Best of Sampled Set (SBOSS) (Castro and Precup (2010)) is a Bayesian RL algorithm which relies on the assumption that the model is sampled from a Dirichlet distribution. From this assumption, it derives uncertainty bounds on the value of state action pairs. It then uses those bounds to decide how many models to sample from the posterior, and how often the posterior should be updated in order to reduce the computational cost of Bayesian updates. The sampling technique is then used to build a merged MDP, as in Asmuth et al. (2009), and to derive the corresponding optimal action with respect to that MDP. In practice, the number of sampled models is determined dynamically with a parameter ϵ . The re-sampling frequency depends on a parameter δ .

Tested values:

- $\epsilon \in \{1.0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6\}$,
- $\delta \in \{9, 7, 5, 3, 1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6\}$.

5.1.8 BEB

The Bayesian Exploration Bonus (BEB) (Kolter and Ng (2009)) is a Bayesian RL algorithm which builds, at each time-step t , the expected MDP given the current posterior. Before solving this MDP, it computes a new reward function $\rho_{BEB}^{(t)}(x, u, y) = \rho_M(x, u, y) + \frac{\beta}{c_{<x,u,y>}^{(t)}}$, where $c_{<x,u,y>}^{(t)}$ denotes the number of times transition $< x, u, y >$ has been observed at

time-step t . This algorithm solves the mean MDP of the current posterior, in which we replaced $\rho_M(\cdot, \cdot, \cdot)$ by $\rho_{BEB}^{(t)}(\cdot, \cdot, \cdot)$, and applies its optimal policy on the current MDP for one step. The bonus β is a parameter controlling the E/E balance. BEB comes with theoretical guarantees of convergence towards Bayesian optimality.

Tested values:

- $\beta \in \{0.25, 0.5, 1, 1.5, 2, 2.5, 3, 4, 8, 16\}$.

5.1.9 COMPUTATION TIMES VARIANCE

Each algorithm has one or more parameters that can affect the number of sampled transitions from a given state, or the length of each simulation. This, in turn, impacts the computation time requirement at each step. Hence, for some algorithms, no choice of parameters can bring the computation time below or over certain values. In other words, each algorithm has its own range of computation time. Note that, for some methods, the computation time is influenced concurrently by several parameters. We present a qualitative description of how computation time varies as a function of parameters in Table 1.

	Offline phase duration	Online phase duration
Random	Almost instantaneous.	Almost instantaneous.
ϵ-Greedy⁴	Almost instantaneous.	Varies in inverse proportion to ϵ . Can vary a lot from one step to another.
OPPS-DS	Varies proportionally to β .	Varies proportionally to the number of features implied in the selected E/E strategy.
BAMCP⁵	Almost instantaneous.	Varies proportionally to K and <i>depth</i> .
BFS3⁶	Almost instantaneous.	Varies proportionally to K , C and <i>depth</i> .
SBOSS⁷	Almost instantaneous.	Varies in inverse proportion to ϵ and δ . Can vary a lot from one step to another, with a general decreasing tendency.
BEB	Almost instantaneous.	Constant.

Table 1: Influence of the algorithm and their parameters on the offline and online phases duration.

-
4. If a random decision is chosen, the model is not solved.
 5. K defines the number of nodes to develop at each step, and *depth* defines the maximal depth of the tree.
 6. K defines the number of nodes to develop at each step, C the branching factor of the tree and *depth* its maximal depth.
 7. The number of models sampled is inversely proportional to ϵ , while the frequency at which the models are sampled is inversely proportional to δ . When an MDP has been sufficiently explored, the number of models to sample and the frequency of the sampling will decrease.

5.2 Benchmarks

In our setting, the transition matrix is the only element which differs between two MDPs drawn from the same distribution. For each $\langle \text{state, action} \rangle$ pair $\langle x, u \rangle$, we define a Dirichlet distribution, which represents the uncertainty about the transitions occurring from $\langle x, u \rangle$. A Dirichlet distribution is parameterised by a set of concentration parameters $\alpha_{\langle x, u \rangle}^{(1)}, \dots, \alpha_{\langle x, u \rangle}^{(n_X)}$.

We gathered all concentration parameters in a single vector θ . Consequently, our MDP distributions are parameterised by ρ_M (the reward function) and several Dirichlet distributions, parameterised by θ . Such a distribution is denoted by $p^{\rho_M, \theta}(\cdot)$. In the Bayesian Reinforcement Learning community, these distributions are referred to as Flat Dirichlet Multinomial distributions (FDMs).

We chose to study two different cases:

- Accurate case: the test distribution is fully known ($p_{\mathcal{M}}^0(\cdot) = p_{\mathcal{M}}(\cdot)$),
- Inaccurate case: the test distribution is unknown ($p_{\mathcal{M}}^0(\cdot) \neq p_{\mathcal{M}}(\cdot)$).

In the inaccurate case, we have no assumption on the transition matrix. We represented this lack of knowledge by a uniform FDM distribution, where each transition has been observed one single time ($\theta = [1, \dots, 1]$).

Sections 5.2.1, 5.2.2 and 5.2.3 describes the three distributions considered for this study.

5.2.1 GENERALISED CHAIN DISTRIBUTION ($p^{\rho^{GC}, \theta^{GC}}(\cdot)$)

The Generalised Chain (GC) distribution is inspired from the five-state chain problem (5 states, 3 actions) (Dearden et al. (1998)). The agent starts at State 1, and has to go through State 2, 3 and 4 in order to reach the last state (State 5), where the best rewards are. The agent has at its disposal 3 actions. An action can either let the agent move from State $x^{(n)}$ to State $x^{(n+1)}$ or force it to go back to State $x^{(1)}$. The transition matrix is drawn from a FDM parameterised by θ^{GC} , and the reward function is denoted by ρ^{GC} . More details can be found in Appendix B.1.

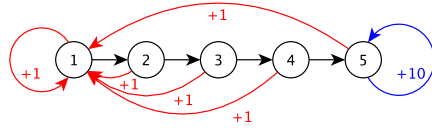


Figure 3: Illustration of the GC distribution.

5.2.2 GENERALISED DOUBLE-LOOP DISTRIBUTION ($p^{\rho^{GDL}, \theta^{GDL}}(\cdot)$)

The Generalised Double-Loop (GDL) distribution is inspired from the double-loop problem (9 states, 2 actions) (Dearden et al. (1998)). Two loops of 5 states are crossing at State 1,

where the agent starts. One loop is a trap: if the agent enters it, it has no choice to exit but crossing over all the states composing it. Exiting this loop provides a small reward. The other loop is yielding a good reward. However, each action of this loop can either let the agent move to the next state of the loop or force it to return to State 1 with no reward. The transition matrix is drawn from an FDM parameterised by θ^{GDL} , and the reward function is denoted by ρ^{GDL} . More details can be found in Appendix B.2.

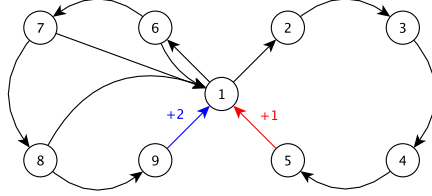


Figure 4: Illustration of the GDL distribution.

5.2.3 GRID DISTRIBUTION ($p^{\rho^{Grid}, \theta^{Grid}}(\cdot)$)

The Grid distribution is inspired from the Dearden’s maze problem (25 states, 4 actions) (Dearden et al. (1998)). The agent is placed at a corner of a 5x5 grid (the **S** cell), and has to reach the opposite corner (the **G** cell). When it succeeds, it returns to its initial state and receives a reward. The agent can perform 4 different actions, corresponding to the 4 directions (up, down, left, right). However, depending on the cell on which the agent is, each action has a certain probability to fail, and can prevent the agent to move in the selected direction. The transition matrix is drawn from an FDM parameterised by θ^{Grid} , and the reward function is denoted by ρ^{Grid} . More details can be found in Appendix B.3.

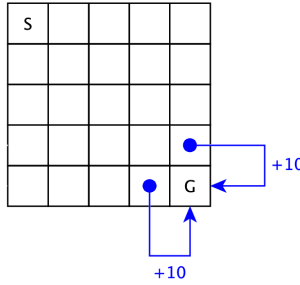


Figure 5: Illustration of the Grid distribution.

5.3 Discussion of the results

5.3.1 ACCURATE CASE

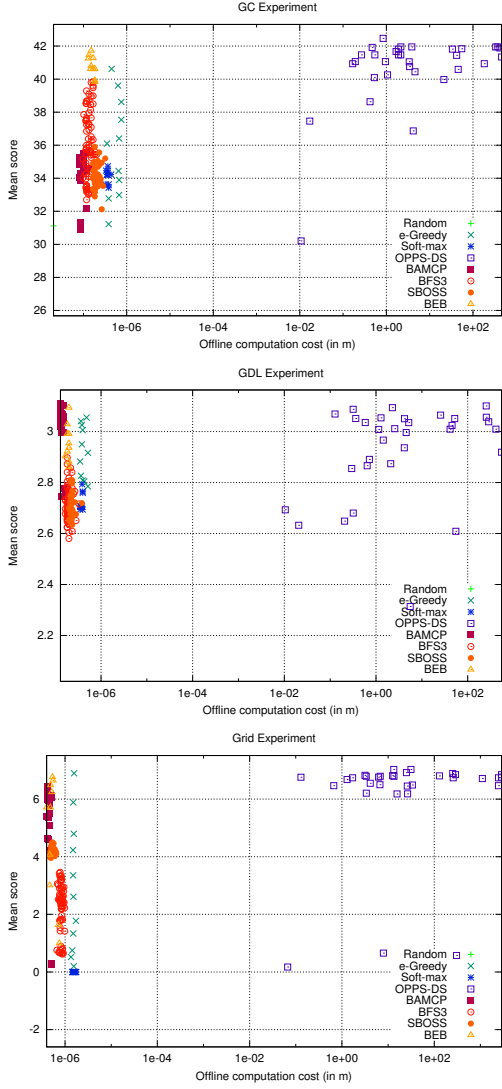


Figure 6: Offline computation cost Vs. Performance

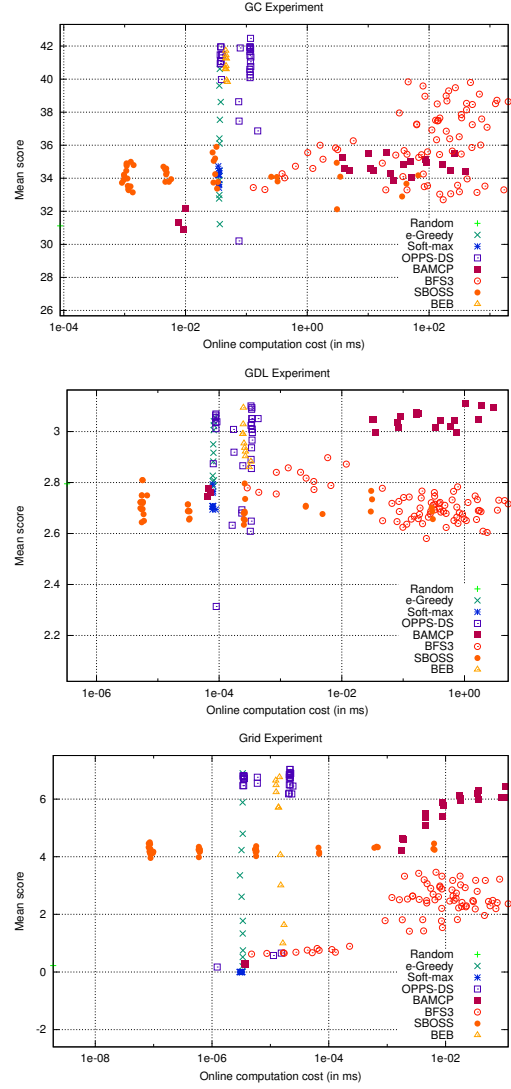
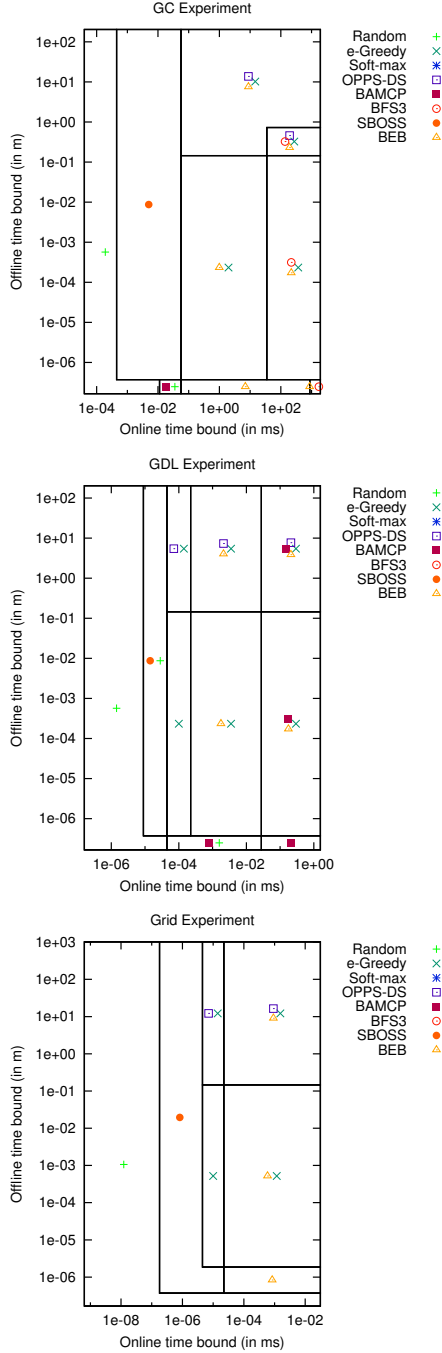


Figure 7: Online computation cost Vs. Performance


 Figure 8: Best algorithms w.r.t of-
fline/online time periods

GC Experiment

Agent	Score
Random	31.12 ± 0.9
e-Greedy ($\epsilon = 0$)	40.62 ± 1.55
Soft-Max ($\tau = 0.1$)	34.73 ± 1.74
OPPS-DS ($Q_2(x, u)/Q_0(x, u)$)	42.47 ± 1.91
BAMCP ($K = 2500$, $depth = 15$)	35.56 ± 1.27
BFS3 ($K = 500$, $C = 15$, $depth = 15$)	39.84 ± 1.74
SBOSS ($\epsilon = 0.001$, $\delta = 7$)	35.9 ± 1.89
BEB ($\beta = 2.5$)	41.72 ± 1.63

GDL Experiment

Agent	Score
Random	2.79 ± 0.07
e-Greedy ($\epsilon = 0.1$)	3.05 ± 0.07
Soft-Max ($\tau = 0.1$)	2.79 ± 0.1
OPPS-DS ($\max(Q_0(x, u), Q_2(x, u))$)	3.1 ± 0.07
BAMCP ($K = 10000$, $depth = 15$)	3.11 ± 0.07
BFS3 ($K = 1$, $C = 15$, $depth = 25$)	2.9 ± 0.07
SBOSS ($\epsilon = 1$, $\delta = 1$)	2.81 ± 0.1
BEB ($\beta = 0.5$)	3.09 ± 0.07

Grid Experiment

Agent	Score
Random	0.22 ± 0.06
e-Greedy ($\epsilon = 0$)	6.9 ± 0.31
Soft-Max ($\tau = 0.05$)	0 ± 0
OPPS-DS ($Q_0(x, u) + Q_2(x, u)$)	7.03 ± 0.3
BAMCP ($K = 25000$, $depth = 15$)	6.43 ± 0.3
BFS3 ($K = 500$, $C = 15$, $depth = 50$)	3.46 ± 0.23
SBOSS ($\epsilon = 0.1$, $\delta = 7$)	4.5 ± 0.33
BEB ($\beta = 0.5$)	6.76 ± 0.3

 Figure 9: Best algorithms w.r.t Perfor-
mance

As it can be seen in Figure 6, OPPS is the only algorithm whose offline time cost varies. In the three different settings, OPPS can be launched after a few seconds, but behaves very poorly. However, its performances increased very quickly when given at least one minute of computation time. Algorithms that do not use offline computation time have a wide range of different scores. This variance represents the different possible configurations for these algorithms, which only lead to different online computation time.

On Figure 7, BAMCP, BFS3 and SBOSS have variable online time costs. BAMCP behaved poorly on the first experiment, but obtained the best score on the second one and was pretty efficient on the last one. BFS3 was good only on the second experiment. SBOSS was never able to get a good score in any cases. Note that OPPS online time cost varies slightly depending on the formula’s complexity.

If we take a look at the top-right point in Figure 8, which defines the less restrictive bounds, we notice that OPPS-DS and BEB were always the best algorithms in every experiment. ϵ -Greedy was a good candidate in the two first experiments. BAMCP was also a very good choice except for the first experiment. On the contrary, BFS3 and SBOSS were only good choices in the first experiment.

If we look closely, we can notice that OPPS-DS was always one of the best algorithm since we have met its minimal offline computation time requirements.

Moreover, when we place our offline-time bound right under OPPS-DS minimal offline time cost, we can see how the top is affected from left to right:

GC: (Random), (SBOSS), (BEB, ϵ -Greedy), (BEB, BFS3, ϵ -Greedy),

GDL: (Random), (Random, SBOSS), (ϵ -Greedy), (BEB, ϵ -Greedy),
(BAMCP, BEB, ϵ -Greedy),

Grid: (Random), (SBOSS), (ϵ -Greedy), (BEB, ϵ -Greedy).

We can clearly see that SBOSS was the first algorithm to appear on the top, with a very small online computation cost, followed by ϵ -Greedy and BEB. Beyond a certain online time bound, BFS3 emerged in the first experiment while BAMCP emerged in the second experiment. Neither of them was able to compete with BEB or ϵ -Greedy in the last experiment.

Soft-max was never able to reach the top regardless the configuration.

Figure 9 reports the best score observed for each algorithm, disassociated from any time measure. Note that the variance is very similar for all algorithms in GDL and Grid experiments. On the contrary, the variance oscillates between 1.0 and 2.0. However, OPPS seems to be the less stable algorithm in the three cases.

5.3.2 INACCURATE CASE

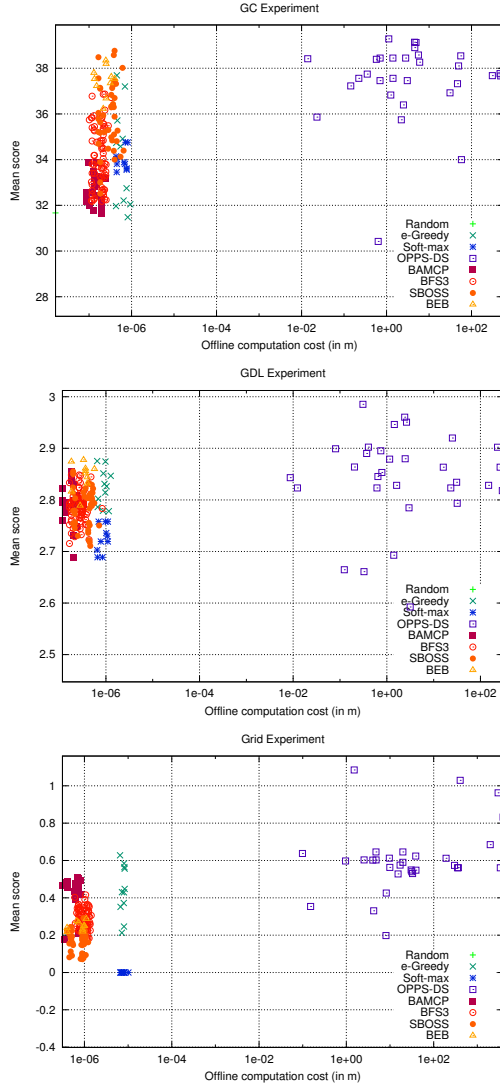


Figure 10: Offline computation cost Vs. Performance

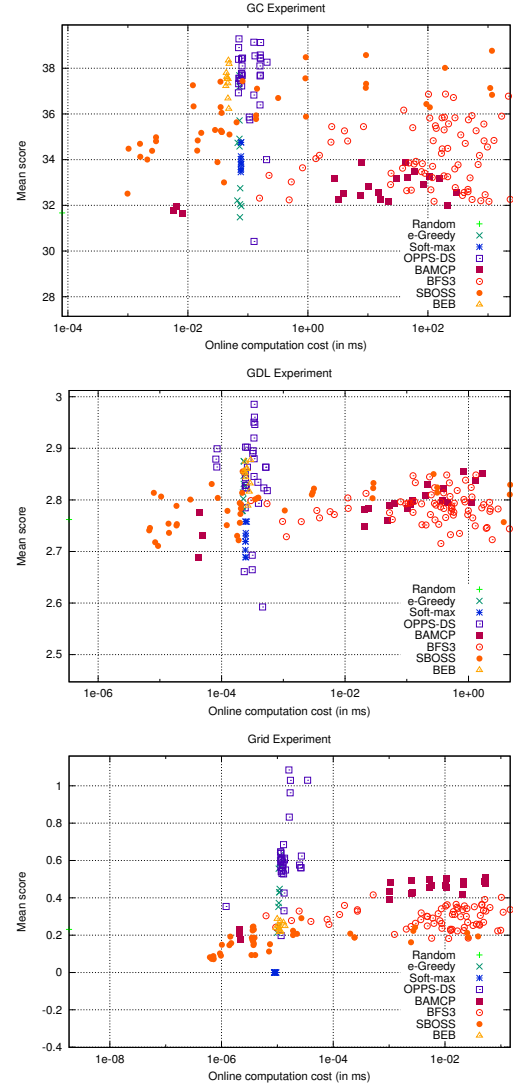


Figure 11: Online computation cost Vs. Performance

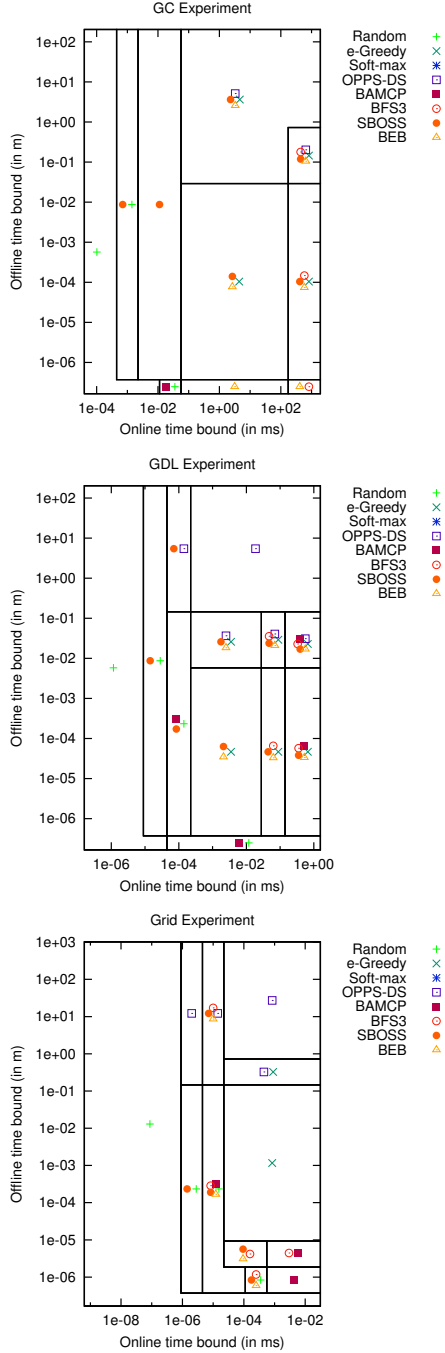


Figure 12: Best algorithms w.r.t of-fline/online time periods

GC Experiment

Agent	Score
Random	31.67 ± 1.05
e-Greedy ($\epsilon = 0$)	37.69 ± 1.75
Soft-Max ($\tau = 0.33$)	34.75 ± 1.64
OPPS-DS ($Q_0(x, u)$)	39.29 ± 1.71
BAMCP ($K = 1250$, $depth = 25$)	33.87 ± 1.26
BFS3 ($K = 1250$, $C = 15$, $depth = 25$)	36.87 ± 1.82
SBOSS ($\epsilon = 1e-06$, $\delta = 0.0001$)	38.77 ± 1.89
BEB ($\beta = 16$)	38.34 ± 1.62

GDL Experiment

Agent	Score
Random	2.76 ± 0.08
e-Greedy ($\epsilon = 0.3$)	2.88 ± 0.07
Soft-Max ($\tau = 0.05$)	2.76 ± 0.1
OPPS-DS ($\max(Q_0(x, u), Q_1(x, u))$)	2.99 ± 0.08
BAMCP ($K = 10000$, $depth = 50$)	2.85 ± 0.07
BFS3 ($K = 1250$, $C = 15$, $depth = 50$)	2.85 ± 0.07
SBOSS ($\epsilon = 0.1$, $\delta = 0.001$)	2.86 ± 0.07
BEB ($\beta = 2.5$)	2.88 ± 0.07

Grid Experiment

Agent	Score
Random	0.23 ± 0.06
e-Greedy ($\epsilon = 0.2$)	0.63 ± 0.09
Soft-Max ($\tau = 0.05$)	0 ± 0
OPPS-DS ($Q_1(x, u) + Q_2(x, u)$)	1.09 ± 0.17
BAMCP ($K = 25000$, $depth = 25$)	0.51 ± 0.09
BFS3 ($K = 1$, $C = 15$, $depth = 50$)	0.42 ± 0.09
SBOSS ($\epsilon = 0.001$, $\delta = 0.1$)	0.29 ± 0.07
BEB ($\beta = 0.25$)	0.29 ± 0.05

Figure 13: Best algorithms w.r.t Performance

As seen in the accurate case, Figure 10 also shows impressive performances for OPPS-DS, which has beaten all other algorithms in every experiment. We can also notice that, as observed in the accurate case, in the Grid experiment, the OPPS-DS agents scores are very close. However, only a few were able to significantly surpass the others, contrary to the accurate case where most OPPS-DS agents were very good candidates.

Surprisingly, SBOSS was a very good alternative to BAMCP and BFS3 in the two first experiments as shown in Figure 11. It was able to surpass both algorithms on the first one while being very close to BAMCP performances in the second. Relative performances of BAMCP and BFS3 remained the same in the inaccurate case, even if the BAMCP advantage is less visible in the second experiment. BEB was no longer able to compete with OPPS-DS and was even beaten by BAMCP and BFS3 in the last experiment. ϵ -Greedy was still a decent choice except in the first experiment. As observed in the accurate case, Soft-max was very bad in every case.

In Figure 12, if we take a look at the top-right point, we can see OPPS-DS is the best choice in the second and third experiment. BEB, SBOSS and ϵ -Greedy share the first place with OPPS-DS in the first one.

If we place our offline-time bound right under OPPS-DS minimal offline time cost, we can see how the top is affected from left to right:

GC: (Random), (Random, SBOSS), (SBOSS), (BEB, SBOSS, ϵ -Greedy),
(BEB, BFS3, SBOSS, ϵ -Greedy),

GDL: (Random), (Random, SBOSS), (BAMCP, Random, SBOSS),
(BEB, SBOSS, ϵ -Greedy), (BEB, BFS3, SBOSS, ϵ -Greedy),
(BAMCP, BEB, BFS3, SBOSS, ϵ -Greedy),

Grid: (Random), (Random, SBOSS), (BAMCP, BEB, BFS3, Random, SBOSS),
(ϵ -Greedy).

SBOSS is again the first algorithm to appear in the rankings. ϵ -Greedy is the only one which could reach the top in every case, even when facing BAMCP and BFS3 fed with high online computation cost. BEB no longer appears to be undeniably better than the others. Besides, the two first experiments show that most algorithms obtained similar results, except for BAMCP which does not appear on the top in the first experiment. In the last experiment, ϵ -Greedy succeeded to beat all other algorithms.

Figure 13 does not bring us more information than those we observed in the accurate case.

5.3.3 SUMMARY

In the accurate case, OPPS-DS was always among the best algorithms, at the cost of some offline computation time. When the offline time budget was too constrained for OPPS-DS, different algorithms were suitable depending on the online time budget:

- **Low online time budget:** SBOSS was the fastest algorithm to make better decisions than a random policy.

- **Medium online time budget**⁸: BEB reached performances similar to OPPS-DS on each experiment.
- **High online time budget**⁹: In the first experiment, BFS3 managed to catch up BEB and OPPS-DS when given sufficient time. In the second experiment, it was BAMCP which has achieved this result. Neither BFS3 nor BAMCP was able to compete with BEB and OPPS-DS in the last experiment.

The results obtained in the inaccurate case were very interesting. BEB was not as good as it seemed to be in the accurate case, while SBOSS improved significantly compared to the others. For its part, OPPS-DS obtained the best overall results in the inaccurate case by outperforming all the other algorithms in two out of three experiments while remaining among the best ones in the last experiment.

6. Conclusion

We have proposed a new extensive BRL comparison methodology which takes into account both performance and time requirements for each algorithm. In particular, our benchmarking protocol shows that no single algorithm dominates all other algorithms on all scenarios. The protocol we introduced can compare any time algorithm to non-anytime algorithms while measuring the impact of inaccurate offline training. By comparing algorithms on large sets of problems, we avoid over fitting to a single problem. Our methodology is associated with an open-source library, BBRL, and we hope that it will help other researchers to design algorithms whose performances are put into perspective with computation times, that may be critical in many applications. This library is specifically designed to handle new algorithms easily, and is provided with a complete and comprehensive documentation website.

Acknowledgments

Michaël Castronovo acknowledges the financial support of the FRIA. Raphael Fonteneau is a postdoctoral fellow of the F.R.S.-FNRS (Belgian Funds for Scientific Research).

8. ± 100 times more than the low online time budget

9. ± 100 times more than the medium online time budget

Appendix A. Pseudo-code of the algorithms

Algorithm 1 ϵ -Greedy

```

1: procedure OFFLINE-LEARNING( $p_{\mathcal{M}}^0(\cdot)$ )
2:    $\hat{M} \leftarrow$  “Build an initial model based on  $p_{\mathcal{M}}^0(\cdot)$ ”
3: end procedure
4:
5: function SEARCH( $x, h$ )
6:   {Draw a random value in  $[0; 1]$ }
7:    $r \leftarrow \mathcal{U}(0, 1)$ 
8:
9:   if  $r < \epsilon$  then {Random case}
10:    return “An action selected randomly”
11:
12:   else {Greedy case}
13:      $\pi_{\hat{M}}^* \leftarrow \text{VALUE-ITERATION}(\hat{M})$ 
14:    return  $\pi_{\hat{M}}^*(x)$ 
15:   end if
16: end function
17:
18: procedure ONLINE-LEARNING( $x, u, y, r$ )
19:   “Update model  $\hat{M}$  w.r.t. transition  $\langle x, u, y, r \rangle$ ”
20: end procedure

```

Algorithm 2 Soft-max

```

1: procedure OFFLINE-LEARNING( $p_{\mathcal{M}}^0(\cdot)$ )
2:    $\hat{M} \leftarrow$  “Build an initial model based on  $p_{\mathcal{M}}^0(\cdot)$ ”
3: end procedure
4:
5: function SEARCH( $x, h$ )
6:   {Draw a random value in  $[0; 1]$ }
7:    $r \leftarrow \mathcal{U}(0, 1)$ 
8:
9:   {Select an action randomly, with a probability proportional to  $Q_{\hat{M}}^*(x, u)$ }
10:   $Q_{\hat{M}}^* \leftarrow$  “Compute the optimal Q-function of  $\hat{M}$ ”
11:  for  $1 \leq i \leq |U|$  do
12:    if  $r < \sum_{j \leq i} \frac{\exp(Q_{\hat{M}}^*(x, u^{(j)})/\tau)}{\sum_{u'} \exp(Q_{\hat{M}}^*(x, u')/\tau)}$  then
13:      return  $u^{(i)}$ 
14:    end if
15:  end for
16: end function
17:
18: procedure ONLINE-LEARNING( $x, u, y, r$ )
19:   “Update model  $\hat{M}$  w.r.t. transition  $\langle x, u, y, r \rangle$ ”
20: end procedure

```

Algorithm 3 OPPS-DS

```

1: procedure OFFLINE-LEARNING( $p_{\mathcal{M}}^0(\cdot)$ )
2:   {Initialise the  $k$  arms of UCB1}
3:   for  $1 \leq i \leq k$  do
4:      $M \sim p_{\mathcal{M}}^0(\cdot)$ 
5:      $R_M^{\pi_i} \leftarrow$  “Simulate strategy  $\pi_i$  on MDP  $M$  over a single trajectory”
6:      $\mu(i) \leftarrow R_M^{\pi_i}$ 
7:      $\theta(i) \leftarrow 1$ 
8:   end for
9:
10:  {Run UCB1 with a budget of  $\beta$ }
11:  for  $k+1 \leq b \leq \beta$  do
12:     $a \leftarrow \arg \max_{a'} \mu(a') + \sqrt{\frac{2 \log(b)}{\theta(a' )}}$ 
13:     $M \sim p_{\mathcal{M}}^0(\cdot)$ 
14:     $R_M^{\pi_a} \leftarrow$  “Simulate strategy  $\pi_a$  on MDP  $M$  over a single trajectory”
15:     $\mu(a) \leftarrow \frac{\theta(a)\mu(a) + R_M^{\pi_a}}{\theta(a)+1}$ 
16:     $\theta(a) \leftarrow \theta(a) + 1$ 
17:  end for
18:
19:  {Select the E/E strategy associated to the most drawn arm}
20:   $a^* \leftarrow \arg \max_{a'} \theta(a')$ 
21:   $\pi_{OPPS} \leftarrow \pi_{a^*}$ 
22: end procedure
23:
24: function SEARCH( $x, h$ )
25:   return  $u \sim \pi_{OPPS}(x, h)$ 
26: end function
27:
28: procedure ONLINE-LEARNING( $x, u, y, r$ )
29:   “Update strategy  $\pi_{OPPS}$  w.r.t. transition  $\langle x, u, y, r \rangle$ ”
30: end procedure

```

Algorithm 4 BAMCP (1/2)

```

1: function SEARCH( $x, h$ )
2:   {Develop a MCTS and compute  $Q(., .)$ }
3:   for  $1 \leq k \leq K$  do
4:      $M \sim p_{\mathcal{M}}^h$ 
5:     SIMULATE( $\langle x, h \rangle, M, 0$ )
6:   end for
7:
8:   {Return the best action w.r.t.  $Q(., .)$ }
9:   return  $\arg \max_u Q(\langle x, h \rangle, u)$ 
10: end function
11:
12: function SIMULATE( $\langle x, h \rangle, M, d$ )
13:   if  $N(\langle x, h \rangle) = 0$  then {New node reached}
14:     “Initialise  $N(\langle x, h \rangle, u)$ ,  $Q(\langle x, h \rangle, u)$ ”
15:      $u \sim \pi_0(\langle x, h \rangle)$ 
16:     “Sample  $x', r$  from model  $M$ ”
17:
18:     {Estimate the score of this node by using the rollout policy}
19:      $R \leftarrow r + \gamma \text{ROLLOUT}(\langle x', hu x' \rangle, P, d)$ 
20:
21:     “Update  $N(\langle x, h \rangle)$ ,  $N(\langle x, h \rangle, u)$ ,  $Q(\langle x, h \rangle, u)$ ”
22:     return  $R$ 
23:   end if
24:
25:   {Select the next branch to explore}
26:    $u \leftarrow \arg \max_{u'} Q(\langle x, h \rangle, u) + c\sqrt{\frac{\log(N(\langle x, h \rangle))}{N(\langle x, h \rangle, u')}})$ 
27:   “Sample  $x', r$  from model  $M$ ”
28:
29:   {Follow the branch and evaluate it}
30:    $R \leftarrow r + \gamma \text{SIMULATE}(\langle x', hu x' \rangle, M, d + 1)$ 
31:
32:   “Update  $N(\langle x, h \rangle)$ ,  $N(\langle x, h \rangle, u)$ ,  $Q(\langle x, h \rangle, u)$ ”
33:   return  $R$ 
34: end function

```

Algorithm 5 BAMCP (2/2)

```

1: procedure ROLLOUT( $\langle x, h \rangle, M, d$ )
2:   if  $\gamma^d R_{max} < \epsilon$  then {Truncate the trajectory if precision  $\epsilon$  has been
   reached}
3:     return 0
4:   end if
5:
6:   {Use the rollout policy to choose the action to perform}
7:    $u \sim \pi_0(x, h)$ 
8:
9:   {Simulate a single transition from  $M$  and continue the rollout process}
10:   $y \sim P_M$ 
11:   $r \leftarrow \rho_M(x, u, y)$ 
12:  return  $r + \gamma$  ROLLOUT( $\langle y, huy \rangle, M, d + 1$ )
13: end procedure
14:
15: procedure ONLINE-LEARNING( $x, u, y, r$ )
16:   "Update the posterior w.r.t. transition  $\langle x, u, y, r \rangle$ "
17: end procedure

```

Algorithm 6 BFS3

```

1: function SEARCH( $x, h$ )
2:   {Update the current Q-function}
3:    $M_{mean} \leftarrow$  "Compute the mean MDP of  $p_{\mathcal{M}}^t(\cdot)$ ."
4:   for all  $u \in \mathcal{U}$  do
5:     for  $1 \leq i \leq C$  do
6:       {Draw  $y$  and  $r$  from the mean MDP of the posterior}
7:        $y \sim P_{M_{mean}}$ 
8:        $r \leftarrow \rho_M(x, u, y)$ 
9:
10:      {Update the Q-value in  $(x, u)$  by using FSSS algorithm}
11:       $Q(x, u) \leftarrow Q(x, u) + \frac{1}{C} [r + \gamma \text{FSSS}(y, d, t)]$ 
12:    end for
13:  end for
14:
15:  {Return the action  $u$  with the maximal Q-value in  $x$ }
16:  return  $\arg \max_u Q(x, u)$ 
17: end function

```

Algorithm 7 FSSS (1/2)

```

1: function FSSS( $x, d, t$ )
2:   {Develop a MCTS and compute bounds on  $V(x)$ }
3:   for  $1 \leq i \leq t$  do
4:     ROLLOUT( $s, d, 0$ )
5:   end for
6:
7:   {Make an optimistic estimation of  $V(x)$ }
8:    $\hat{V}(x) \leftarrow \max_u U_d(x, u)$ 
9:   return  $\hat{V}(x)$ 
10: end function

```

Algorithm 8 FSSS (2/2)

```

1: procedure ROLLOUT( $x, d, l$ )
2:   if  $d = l$  then {Stop when reaching the maximal depth}
3:     return
4:   end if
5:
6:   if  $\neg \text{Visited}_d(x)$  then {New node reached}
7:     {Initialise this node}
8:     for all  $u \in U$  do
9:       "Initialise  $N_d(x, u, x'), R_d(x, u)$ "
10:      for  $1 \leq i \leq C$  do
11:        "Sample  $x', r$  from  $M$ "
12:        "Update  $N_d(x, u, x'), R_d(x, u)$ "
13:
14:        if  $\neg \text{Visited}_d(x')$  then
15:           $U_{d+1}(x'), L_{d+1}(x') = V_{max}, V_{min}$ 
16:        end if
17:      end for
18:    end for
19:
20:    {Back-propagate this node's information}
21:    BELLMAN-BACKUP( $x, d$ )
22:
23:     $\text{Visited}_d(x) \leftarrow \text{true}$ 
24:  end if
25:
26:  {Select an action and simulate a transition optimistically}
27:   $u \leftarrow \arg \max_u U_d(x, u)$ 
28:   $x' \leftarrow \arg \max_{x'} (U_{d+1}(x') - L_{d+1}(x')) N_d(x, u, x')$ 
29:
30:  {Continue the rollout process and back-propagate the result}
31:  ROLLOUT( $x', d, l + 1$ )
32:  BELLMAN-BACKUP( $x, d$ )
33:  return
34: end procedure

```

Algorithm 9 SBOSS (1/2)

```

1: function SEARCH( $x, h$ )
2:   {Compute the transition matrix of the mean MDP of the posterior}
3:    $M_{mean} \leftarrow$  "Compute the mean MDP of  $p_{\mathcal{M}}^t(\cdot)$ ."
4:    $P_t \leftarrow P_{M_{mean}}$ 
5:
6:   {Update the policy to follow if necessary}
7:    $\forall (x, u) : \Delta(x, u) = \sum_{y \in X} \frac{|P_t(x, u, y) - P_{lastUpdate}(x, u, y)|}{\sigma(x, u, y)}$ 
8:   if  $t = 1$  or  $\exists (x', u') : \Delta(x', u') > \delta$  then
9:     {Sample some transition vectors for each state-action pair}
10:     $S \leftarrow \{\}$ 
11:    for all  $(x, u) \in X \times U$  do
12:      {Compute the number of transition vectors to sample for  $(x, u)$ }
13:       $K_t(x, u) \leftarrow \max_y \left\lceil \frac{\sigma^2(x, u, y)}{\epsilon} \right\rceil$ 
14:
15:      {Sample  $K_t(x, u)$  transition vectors from  $\langle x, u \rangle$ , sampled from
the posterior}
16:      for  $1 \leq k \leq K_t(x, u)$  do
17:         $S \leftarrow S \cup$  "A transition vector from  $\langle x, u \rangle$ , sampled from the
posterior"
18:      end for
19:    end for
20:
21:     $M^\# \leftarrow$  "Build a new MDP by merging all transitions from  $S$ "
22:     $\pi_{M^\#}^* \leftarrow$  VALUE-ITERATION( $M^\#$ )
23:     $\pi_{SBOSS} \leftarrow$  FIT-ACTION-SPACE( $\pi_{M^\#}^*$ )
24:     $P_{lastUpdate} \leftarrow P_t$ 
25:  end if
26:
27:  {Return the optimal action in  $x$  w.r.t.  $\pi_{SBOSS}$ }
28:  return  $u \sim \pi_{SBOSS}(x)$ 
29: end function

```

Algorithm 10 SBOSS (2/2)

```

1: function FIT-ACTION-SPACE( $\pi_{M\#}^*$ )
2:   for all  $x \in X$  do
3:      $\pi(x) \leftarrow \pi_{M\#}^*(x) \bmod |U|$ 
4:   end for
5:
6:   return  $\pi$ 
7: end function
8:
9: procedure ONLINE-LEARNING( $x, u, y, r$ )
10:  “Update the posterior w.r.t. transition  $\langle x, u, y, r \rangle$ ”
11: end procedure
    
```

Algorithm 11 BEB

```

1: procedure SEARCH( $x, h$ )
2:   $M \leftarrow$  “Compute the mean MDP of  $p_{\mathcal{M}}^t(\cdot)$ .”
3:
4:  {Add a bonus reward to all transitions}
5:  for  $\langle x, u, y \rangle \in \mathcal{X} \times \mathcal{U} \times \mathcal{X}$  do  $\rho_M(x, u, y) \leftarrow \rho_M(x, u, y) + \frac{\beta}{c_{\langle x, u, y \rangle}^{(t)}}$ 
6:  end for
7:
8:  {Compute the optimal policy of the modified MDP}
9:   $\pi_M^* \leftarrow$  VALUE-ITERATION( $M$ )
10:
11:  {Return the optimal action in  $x$  w.r.t.  $\pi_M^*$ }
12:  return  $u \sim \pi_M^*(x)$ 
13: end procedure
14:
15: procedure ONLINE-LEARNING( $x, u, y, r$ )
16:  “Update the posterior w.r.t. transition  $\langle x, u, y, r \rangle$ ”
17: end procedure
    
```

Appendix B. MDP distributions in detail

In this section, we describe the MDPs drawn from the considered distributions in more detail. In addition, we also provide a formal description of the corresponding θ (parameterising the FDM used to draw the transition matrix) and ρ_M (the reward function).

B.1 Generalised Chain distribution

On those MDPs, we can identify two possibly optimal behaviours:

- The agent tries to move along the chain, reaches the last state, and collect as many rewards as possible before returning to State 1;

- The agent gives up to reach State 5 and tries to return to State 1 as often as possible.

B.1.1 FORMAL DESCRIPTION

$$X = \{1, 2, 3, 4, 5\}, U = \{1, 2, 3\}$$

$$\forall u \in U :$$

$$\theta_{1,u}^{GC} = [1, 1, 0, 0, 0]$$

$$\theta_{2,u}^{GC} = [1, 0, 1, 0, 0]$$

$$\theta_{3,u}^{GC} = [1, 0, 0, 1, 0]$$

$$\theta_{4,u}^{GC} = [1, 0, 0, 0, 1]$$

$$\theta_{5,u}^{GC} = [1, 1, 0, 0, 1]$$

$$\forall x, u \in X \times U :$$

$$\rho^{GC}(x, u, 1) = 2.0$$

$$\rho^{GC}(x, u, 5) = 10.0$$

$$\rho^{GC}(x, u, y) = 0.0, \forall y \in X \setminus \{1, 5\}$$

B.2 Generalised Double-Loop distribution

Similarly to the GC distribution, we can also identify two possibly optimal behaviours:

- The agent enters the “good” loop and tries to stay in it until the end;
- The agent gives up and chooses to enter the “bad” loop as frequently as possible.

B.2.1 FORMAL DESCRIPTION

$$X = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}, U = \{1, 2\}$$

$$\forall u \in U :$$

$$\theta_{1,u}^{GDL} = [0, 1, 0, 0, 0, 1, 0, 0, 0]$$

$$\theta_{2,u}^{GDL} = [0, 0, 1, 0, 0, 0, 0, 0, 0]$$

$$\theta_{3,u}^{GDL} = [0, 0, 0, 1, 0, 0, 0, 0, 0]$$

$$\theta_{4,u}^{GDL} = [0, 0, 0, 0, 1, 0, 0, 0, 0]$$

$$\theta_{5,u}^{GDL} = [1, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$\theta_{6,u}^{GDL} = [1, 0, 0, 0, 0, 0, 1, 0, 0]$$

$$\theta_{7,u}^{GDL} = [1, 0, 0, 0, 0, 0, 0, 1, 0]$$

$$\theta_{8,u}^{GDL} = [1, 0, 0, 0, 0, 0, 0, 0, 1]$$

$$\theta_{9,u}^{GDL} = [1, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$\forall u \in U :$$

$$\rho^{GDL}(5, u, 1) = 1.0$$

$$\rho^{GDL}(9, u, 1) = 2.0$$

$$\rho^{GDL}(x, u, y) = 0.0, \forall x \in X, \forall y \in X : y \neq 1$$

B.3 Grid distribution

MDPs drawn from the Grid distribution are 2-dimensional grids. Since the agents considered do not manage multi-dimensional state spaces, the following bijection was defined:

$$\{1, 2, 3, 4, 5\} \times \{1, 2, 3, 4, 5\} \rightarrow X = \{1, 2, \dots, 25\} : n(i, j) = 5(i - 1) + j$$

where i and j are the row and column indexes of the cell on which the agent is.

When the agent reaches the **G** cell (in $(5, 5)$), it is directly moved to $(1, 1)$, and will perceive its reward of 10. In consequence, State $(5, 5)$ is not reachable.

To move inside the Grid, the agent can perform four actions: $U = \{up, down, left, right\}$. Those actions only move the agent to one adjacent cell. However, each action has a certain probability to fail (depending on the cell on which the agent is). In case of failure, the agent does not move at all. Besides, if the agent tries to move out of the grid, it will not move either. Discovering a reliable (and short) path to reach the **G** cell will determine the success of the agent.

B.3.1 FORMAL DESCRIPTION

$$X = \{1, 2, \dots, 25\}, U = \{up, down, left, right\}$$

$$\begin{aligned} \forall (i, j) &\in \{1, 2, 3, 4, 5\} \times \{1, 2, 3, 4, 5\} \\ \forall (k, l) &\in \{1, 2, 3, 4, 5\} \times \{1, 2, 3, 4, 5\} : \end{aligned}$$

$$\theta_{n(i,j),u}^{Grid}(n(i,j)) = 1, \forall u \in U$$

$$\begin{aligned} \theta_{n(i,j),up}^{Grid}(n(i-1,j)) &= 1, (i-1) \geq 1 & \rho^{Grid}((4,5),down,(1,1)) &= 10.0 \\ \theta_{n(i,j),down}^{Grid}(n(i+1,j)) &= 1, (i+1) \leq 5, (i,j) \neq (4,5) & \rho^{Grid}((5,4),right,(1,1)) &= 10.0 \\ \theta_{n(i,j),left}^{Grid}(n(i,j-1)) &= 1, (j-1) \geq 1 & & \\ \theta_{n(i,j),right}^{Grid}(n(i,j+1)) &= 1, (j+1) \leq 5, (i,j) \neq (5,4) & \rho^{Grid}((i,j),u,(k,l)) &= 0.0, \forall u \in U \end{aligned}$$

$$\theta_{n(4,5),down}^{Grid}(n(1,1)) = 1$$

$$\theta_{n(5,4),right}^{Grid}(n(1,1)) = 1$$

$$\theta_{n(i,j),u}^{Grid}(n(k,l)) = 0, else$$

Appendix C. Paired sampled Z-test

Let π_A and π_B be the two agents we want to compare. We played the two agents on the same N MDPs, denoted by M_1, \dots, M_N . Let $R_{M_i}^{\pi_A}$ and $R_{M_i}^{\pi_B}$ be the scores we observed for the two agents on M_i .

Step 1 - Hypothesis

We compute the mean and the standard deviation of the differences between the two sample sets, denoted by \bar{x}_d and \bar{s}_d , respectively.

$$\bar{x}_d = \frac{1}{N} \sum_{i=1}^N R_{M_i}^{\pi_A} - R_{M_i}^{\pi_B}$$

$$\bar{s}_d = \frac{1}{N} \sum_{i=1}^N (\bar{x}_d - (R_{M_i}^{\pi_A} - R_{M_i}^{\pi_B}))^2$$

If $N \geq 30$, \bar{s}_d is a good estimation of σ_d , the standard deviation of the differences between the two populations ($\bar{s}_d \approx \sigma_d$). In other words, σ_d is the standard deviation we should observe when testing the two algorithms on a number of MDPs tending towards infinity. This was always the case in our experiments.

We now set Hypothesis H_0 and Hypothesis H_α :

$$H_0 : \mu_d = 0$$

$$H_\alpha : \mu_d > 0$$

Our goal is to determine if μ_d , the mean of the differences between the two populations, is equal or greater than 0. More expressly, we want to know if the differences between the two agents' performances is significant (H_α is correct) or not (H_0 correct). Only one of those hypotheses can be true.

Step 2 - Test statistic

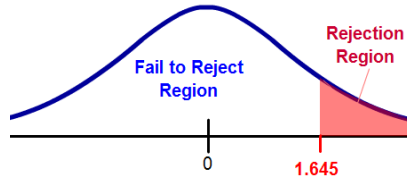
The test statistic consists to compute a certain value Z :

$$Z = \frac{\bar{x}_d}{\sigma_d / \sqrt{N}}$$

This value will help us to determine if we should accept (or reject) hypothesis H_α .

Step 3 - Rejection region

Assuming we want our decision to be correct with a probability of failure of α , we will have to compare Z with Z_α , a value of a Gaussian curve. If $Z > Z_\alpha$, it means we are in the rejection region (R.R.) with a probability equal to $1 - \alpha$. For a confidence of 95%, Z_α should be equal to 1.645.



Being in the R.R. means we have to reject Hypothesis H_0 (and accept Hypothesis H_α). In the other case, we have to accept Hypothesis H_0 (and reject Hypothesis H_α).

Step 4 - Decision

At this point, we have either accepted Hypothesis H_0 or Hypothesis H_α .

- **Accepting Hypothesis H_0 ($Z < Z_\alpha$):** The two algorithms π_A and π_B are not significantly different.
- **Accepting Hypothesis H_α ($Z \geq Z_\alpha$):** The two algorithms π_A and π_B are significantly different. Therefore, the algorithm with the greatest mean is definitely better with 95% confidence.

References

- J. Asmuth and M. Littman. Approaching Bayes-optimality using Monte-Carlo tree search. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011.
- J. Asmuth, L. Li, M.L. Littman, A. Nouri, and D. Wingate. A Bayesian sampling approach to exploration in Reinforcement Learning. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 19–26. AUAI Press, 2009.
- J.Y. Audibert, R. Munos, and C. Szepesvári. Tuning bandit algorithms in stochastic environments. In *Algorithmic Learning Theory*, pages 150–165. Springer, 2007.
- P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.
- P. S. Castro and D. Precup. Smarter sampling in model-based bayesian reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 200–214. Springer, 2010.
- M. Castronovo, F. Maes, R. Fonteneau, and D. Ernst. Learning exploration/exploitation strategies for single trajectory Reinforcement Learning. *Journal of Machine Learning Research (JMLR)*, pages 1–9, 2012.
- M. Castronovo, R. Fonteneau, and D. Ernst. Bayes Adaptive Reinforcement Learning versus Off-line Prior-based Policy Search: an Empirical Comparison. *23rd annual machine learning conference of Belgium and the Netherlands (BENELEARN 2014)*, pages 1–9, 2014.
- R. Dearden, N. Friedman, and S. Russell. Bayesian Q-learning. In *Proceedings of Fifteenth National Conference on Artificial Intelligence (AAAI)*, pages 761–768. AAAI Press, 1998.
- R. Dearden, N. Friedman, and D. Andre. Model based Bayesian exploration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 150–159. Morgan Kaufmann, 1999.
- A. Guez, D. Silver, and P. Dayan. Efficient Bayes-adaptive Reinforcement Learning using sample-based search. In *Neural Information Processing Systems (NIPS)*, 2012.

- M. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.
- L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. *European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
- J. Zico Kolter and Andrew Y. Ng. Near-Bayesian exploration in polynomial time. In *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009.
- M. Strens. A Bayesian framework for Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML)*, pages 943–950. ICML, 2000.